

Design and Implementation of The Java Embedded Micro-kernel Software Architecture

Mark Baker and Hong Ong
The Distributed Systems Group
University of Portsmouth
Portsmouth, PO1 2EG, UK

Mark.Baker@computer.org and Hong.Ong@port.ac.uk

Abstract

The Java Embedded Micro-kernel (JEM) architecture is an object-oriented execution environment for small devices with limited system resources, such as intelligent network cards or PDAs. The core of the architecture is a Java-based micro-kernel providing a framework for dynamically loading service modules onto embedded devices at runtime and mechanisms for interconnecting these service modules through well-defined interfaces. In this paper, we describe the key concepts of the JEM architecture and how they were implemented.

Keywords

Java, JNI, Micro-kernel, Modules, Communication API

1 Introduction

According to research reported by the International Data Corporation (IDC) [1], revenue from embedded operating environment sales is expanding at an annual rate of 30%. The main reason for this growing take up is due to the increasing numbers of microprocessors and other computer-based resources, are being embedded in the modern generation of devices, such as mobile phones, pagers, PDAs, set-top boxes, and other process controllers. Furthermore, devices now are equipped with significantly more hardware resources than their predecessors. For example, the latest generation of network card from Myricom [2] comes with 8 Mbytes of memory and a 200 MHz RISC processor (PCI64C). Equally, PDAs can now be found with 64 Mbytes of memory and a 206 MHz processor [3].

The presence of these additional hardware capabilities presents an opportunity for software engineers to develop and implement sophisticated software to manage and run on these embedded devices. In some cases, it provides the possibility for these devices to off-load services normally undertaken by a traditional computer, for example with intelligent network cards. In particular, there is a great demand for software that can easily incorporate new services and “smart” features for a wide range of embedded products. Clearly, designing reliable software for the increasingly complex and sophisticated embedded devices has become a major challenge.

Essentially, the challenge comes from the software design issues such as heterogeneity, code-reusability, concurrency, and modules interaction. Therefore, it is necessary to have a runtime environment for developing and testing software that specifically caters to embedded devices. This run-time environment must provide the necessary mechanisms by which the modules can be executed and perhaps interact. By modules, we mean a distributed object that is capable of performing a specific task with a set of constraints. Ideally, the modules will be reusable, platform independent, and embody valuable domain-specific expertise.

It is the objective of the JEM project to design and implement a generic runtime environment that can be used in a range of embedded devices – from network cards to PDAs and from mobile phones to printers [4]. For this reason, one of the design goals is to minimize the device

dependent code and thus maximize portability. Java is an excellent choice for the implementation of the JEM architecture as it offers many advantages, including:

- Platform independence that allows code reuse across processors and product lines. This will enable developers to deploy the same applications to a range of devices.
- Applications development can begin on a variety of desktop environments well before the targeted deployment hardware is available.
- Java provides a framework for secure networking, and thus adds the possibility of remote access for configuration and managements of devices.
- The Java class loader permits dynamic loading of classes that will facilities the evolution of the application embedded in the devices.

Central to the JEM architecture is a Java-based micro-kernel, which will provide the necessary mechanisms for the various runtime aspects of the system. In addition, the Java-based micro-kernel will allow a range of service modules, written in Java, to be dynamically loaded at runtime. The Java-based micro-kernel together with the service modules forms the Java Embedded Micro-kernel (JEM) architecture.

This paper presents the implementation of JEM for the Myrinet adaptor. We have used the GNU Java compiler (GCJ) to compile Java source code and byte code into native code of the target device. The JEM is almost entirely written in Java and does not require a JVM. This is possible because all the Java code is being compiled into native code. The rest of this paper is organized as follows. In Section 2, we give an overview of the JEM architecture. Since our initial base level implementation of JEM was to runs directly on Myrinet adaptor, we briefly describe the capability of the Myrinet 2000 hardware in Section 3. In Section 4, we discuss a variety of implementation difficulties and the solutions for solving them. We then discuss and compare the JEM to some related works in Section 5. Finally, in Section 6, we conclude and summarize the benefits and lesson learned. Additionally, we indicate the current state of the implementation and outline several possibilities of future work.

2 An Overview of the JEM Software Architecture

Micro-kernels are not new. System architects have been designing and using micro-kernels since the mid-80s. A micro-kernel attempts to decompose the conventional monolithic kernel horizontally by migrating the kernel functionality into servers running in user space. This decomposition allows the micro-kernel to be small, fast, and does not burden each CPU with services, such as a complete file system, that it does not need [5]. As a consequence, the micro-kernel provides the foundation for modular, scalable and portable extensions. Typically, micro-kernels have been used where there are limited system resources; the ability to load only essential services at any one time has been the best means of reducing kernel overheads [6].

Recently, there has been increasing interest in using Java-based micro-kernels [7] due to the advantages of Java briefly outlined earlier. Using Java, however, is not without problems. Some drawbacks of Java are that it is computationally slower than alternatives such as C and C++, has non-deterministic treatment of asynchronous exception events, and non-bounded resource usage, which are normally needed to meet the constraints of embedded software.

Although some solutions have been proposed to solve these shortcomings of Java regarding its use for embedded software programming [8][9], they have by no means completely eliminated the problems. Meanwhile, modern devices, which have significant memory and a reasonably fast processor, along with technologies such as optimizing compilers, have made Java a realistic option for embedded devices. However, with Java-based systems there is still a need to minimize memory and other resources used. Consequently, we feel that a micro-kernel based system, at this time, is the only viable alternative for providing the required experimental and research platform.

The design of JEM architecture is based largely, but not entirely, on L4 [10] that was originally proposed by Liedtke in the mid 90's. This design decision was made mainly due to the fact that most existing micro-kernels are not "tiny" (with the exception of QNX micro-kernel [11]). Additionally, there are several reference implementations of L4, for example L4/x86 and Fiasco [12], which have reported overcoming the IPC performance problems that most micro-kernels experienced in the early 80's up till the present. Consequently, we felt that instead of re-inventing the wheel of micro-kernel design, it would be more sensible to start a Java-based micro-kernel design based on existing one, i.e., L4.

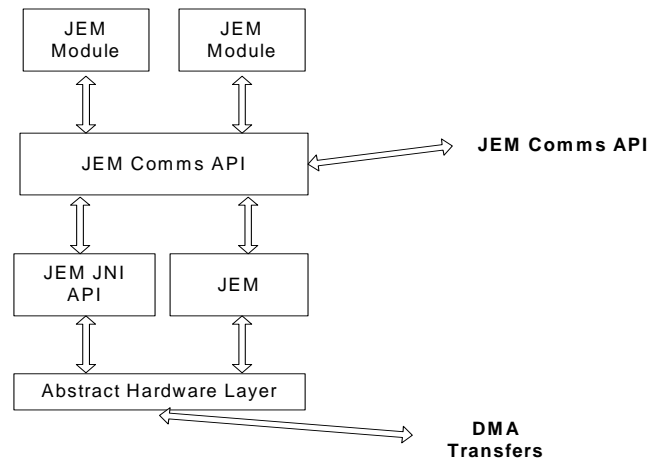


Figure 1: Generic JEM Architecture

Figure 1 shows the generic JEM architecture, this consists of the following parts:

- The Abstract Hardware Layer (AHL) is the JEM interface to the underlying device hardware. It is the means by which JEM interacts with the device processor registers and memory.
- The JEM JNI API is the fast by-pass route for the JEM communications API to interact with the device AHL.
- The JEM communications API is a generic messaging API that provides JEM modules with data I/O services. This API interacts with its peer on other devices.
- The JEM Modules provide application level services for a Java application.
- The JEM is the Java-based micro-kernel (described fully next).

JEM supports three abstractions – modules and threads, address space, and messages. The modules abstraction arises from separating the traditional notion of a process into two sub-concepts. A module contains the resources associated with a process such as address space and I/O or file descriptors. A module does not perform computation itself but serves as a framework in which thread(s) can operate. Hence, a thread is a control unit, which contains the minimal processing state associated with a computation for example a program counter, a stack pointer, and other hardware register state information. Typical examples are an IP router, a Web server, modules that implements the user-level protocols such as VIA [13], a video decoder.

Address space is a mapping associated with each virtual page to a physical page frame. Each module has a virtual address space within which its thread executes. Like L4, JEM supports the recursive construction of address space concept. Depending on the capabilities of the target device, the module could construct and maintain further address spaces on top of the initial space. In order to support this concept, JEM provides three primitive operations: grant, map, and flush. The grant operation allows the owner of the address space to *grant* any of its pages to another space if and only if the recipient agrees. The map operation enables the owner of the address

space to *map* any of its pages into another address space; again, the recipient has to agree. The flush operation allows the owner to de-map any of its pages. By supporting these three primitives, a higher-level memory server could be implemented on top of JEM to offer various memory protection schemes. A more detail explanation of the concept of recursive construction of address space could be found in [10].

All inter and intra communications among modules is carried out in the form of messages. A message is simply considered as a stream of bytes and contains the data to be communicated. This enables JEM to move data efficiently from one address space to another address space.

3 Myrinet 2000 Adaptor

Myrinet-2000, developed by Myricom, is a proprietary network technology and is compliant to the Physical and Data Link layer defined in the ANSI/VITA 26-1998 standard [14]. Myrinet is a switched, Gigabit per second network that is widely used in Beowulf clusters and embedded system. A Myrinet-2000 network is composed of crossbar switches and network adaptors. Network adaptors are connected to the switches through point-to-point duplex links and the crossbar switches can be interconnected in an arbitrary topology.

The Myrinet-2000 network adaptor is a programmable communication device that provides an interface to the System Area Network (SAN). It consists of three major components (see Figure 2):

1. A custom VLSI (LANai) chip,
2. A synchronous static RAM memory,
3. A PCI Bridge and a DMA controller.

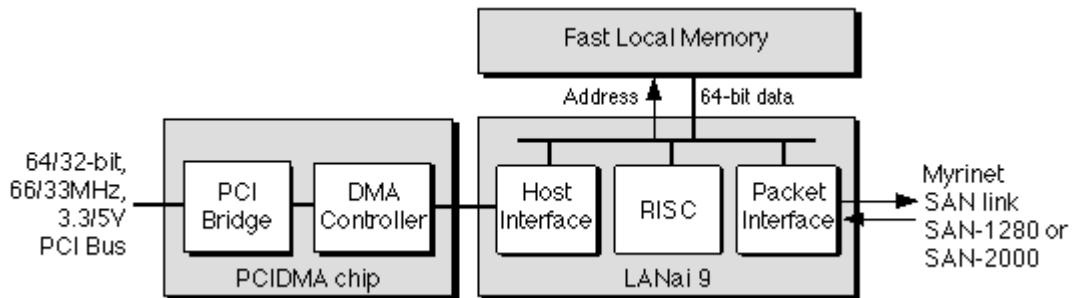


Figure 2: LANai9 Architecture

Like other modern network adaptors, Myrinet adapters have a programmable network interface processor known as LANai9. LANai9 is a 32-bit RISC processor that operates at up to 133MHz for the PCI64B interfaces, or at up to 200MHz for the PCI64C interfaces. Using the Myrinet Control Program (MCP), which is stored in the onboard static RAM (SRAM), LANai9 controls the data transfer between the host and the network (through the host and packet interface), performs data buffer management (through memory interface), and maintains network mapping and monitoring. The benefit of a programmable network processor is that it enables researchers to explore many protocol design options. In our case, this presents an opportunity for us to investigate the possibility of downloading an execution environment to run services.

The onboard Myrinet-2000 SRAM size ranges from 2Mbytes to 8 Mbytes and operates at the same clock speed as LANai9, i.e., at up to 133 MHz for the PCI64B interfaces or at up to 200MHz for the PCI64C interfaces. This suggests that the maximum attainable bandwidth is approximately 1,064 Mbytes/s and 1,600 Mbytes/s for the PCI64B and PCI64C respectively. The SRAM is accessible from both the onboard local bus (LBUS) and the external system bus

(EBUS). LBUS and EBUS are both 64-bit wide but the LBUS is clocked at twice the chip-clock speed, permitting two LBUS memory accesses per clock cycle.

To increase the data transfer rate, a Myrinet-2000 network adaptor is equipped with three DMA engines. Two DMA engines are associated with the packet interface: one for receiving packets and one for sending packets. The third DMA engine is used for data transfer between the SRAM and the host system memory through the host interface. Like most systems that support DMA, the on board memory can be mapped into user space and is thus accessible directly to user processes. This mapping is performed in two steps: the OS first maps the NIC address space into kernel space and then, on a request by user, the kernel region is mapped into user space. This memory mapping technique is commonly known as “memory pinning”. In order to support zero-copy APIs efficiently, the DMA operations can be performed with arbitrary byte counts and byte alignments. Additionally, the DMA engine also computes the IP checksum for each transfer and provides a "doorbell" signaling mechanism that allows the host to write anywhere within the doorbell region, and have the address and data stored in a FIFO queue in the local memory.

The host processor can also access the Myrinet-2000 SRAM through the programmable input/output (PIO) interfaces. With PIO, the host processor reads the data from the host memory and writes it into the Myrinet-2000 SRAM. This mode of data transfer typically results in many PCI I/O bus transactions. Although Myrinet-2000 PCI64 interfaces are capable of sustained PCI data rates approaching the limits of the PCI bus, the network performance greatly depends on the data transfer rate of the host’s memory and PCI-bus implementation.

4 JEM Implementation Issues

The initial base level implementation of the JEM was to runs directly on the Myrinet 2000 network interfaces. Figure 3 shows the interaction of a JEM enabled network card and its host workstation. This particular example shows the communications layers that would be used by a MPJ [15][16][17] application sending/receiving messages to and from another JEM enabled network card.

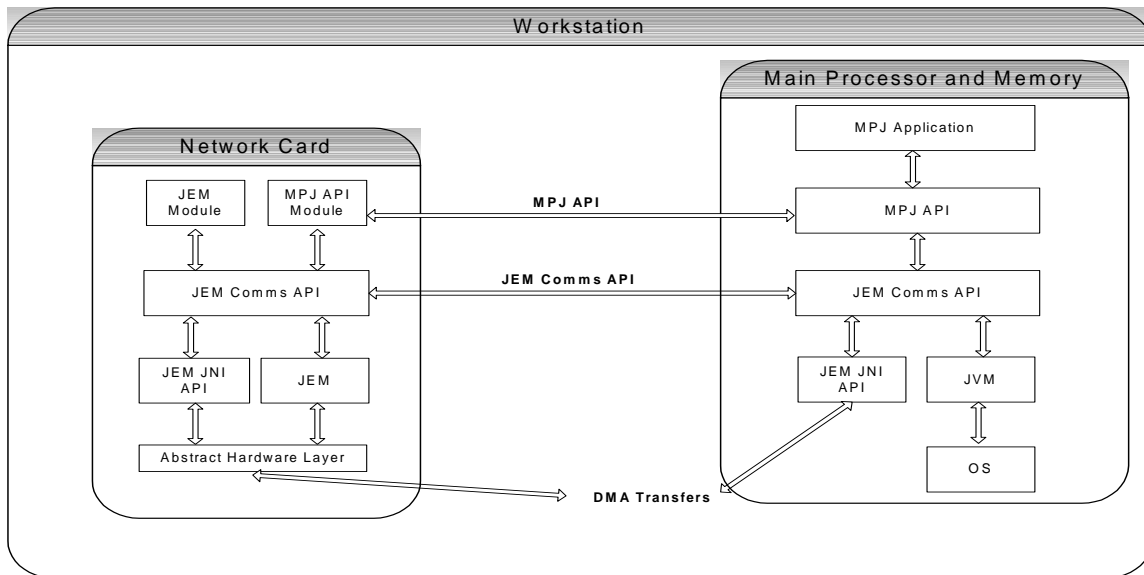


Figure 3: The Relationship between a JEM enabled network card and host workstation

4.1 Implementation strategy

To embed the JEM system into the network card, we used the GNU Java compiler (GCC 2.95) to compile the Java source code into native code. This required us to also customize the loader that was used to push the native code onto the network card. Figure 4 shows the JEM system compilation and loading process.

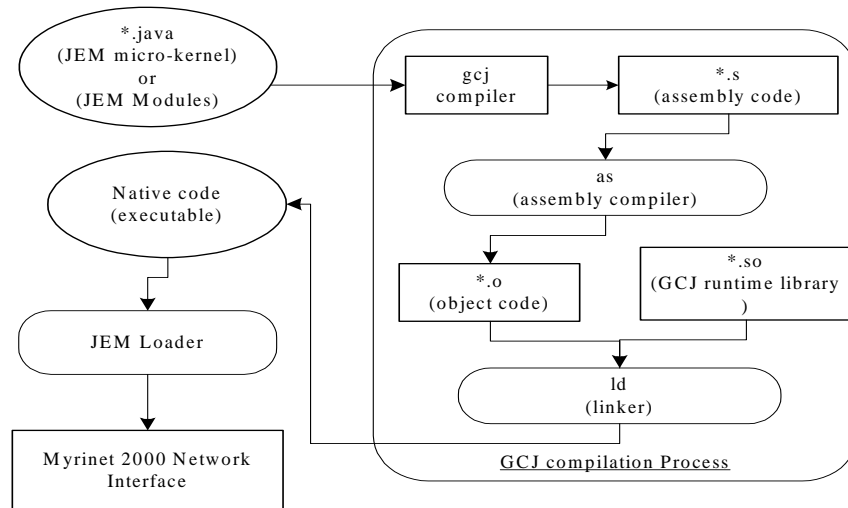


Figure 4: The JEM Compilation Process

The GNU Compiler Collection (GCC) is a suite of compilers for the C, Fortran, C++, Objective C, and Java languages. The strength of the GCC is that all the compilers share the same basic structure, have a common set of back ends and have been ported to various platforms, including a large range of processors used by embedded systems. The final ability will ensure that the JEM and its services modules can be easily ported to a range embedded devices with little or no modification.

Since the design of JEM is largely based on the L4/x86 specification, the JEM implementation is straightforward and just requires mapping the definition of the x86 specification onto the Myrinet network hardware. It should be noted that these mappings do not require the modification of the other L4 system level API definitions. However, despite the simplicity of the API specifications and the compilation and loading process, we encountered several problems that were not foreseen during the design stage. These problems were mainly related the GCJ runtime support and partially linked to the constraints of the Java language specification. For the remainder of this section we highlight some of the more interesting challenges encountered and describe the solutions adopted to resolve them.

4.2 Modification to GCJ runtime support

It is simple to use the GNU C and C++ compilers to build an embedded and stand-alone runtime environment. This is due to the C compiler needing little runtime support, while the C++ runtime support is still relatively modest. However, the Java runtime support is much more complex. This is because the Java language specification explicitly requires the runtime environment to support threads, type reflection, exception handling, garbage collection, and all the primitive methods from the `java.lang` package.

The GCJ addresses these needs by handling a number of runtime classes, including `Object`, `Class`, `Throwable`, `Error`, `Exception`, and `Thread`, in a special way. In particular, the

GCJ silently inserts a number of fields into these classes and disallows any new fields from being defined in the corresponding Java code. To make matter worse, some of the inserted fields have types that cannot be represented in Java. Table 1 shows some of the instance fields silently inserted to the `Object` class and `Class` class.

Field name	Field type
<code>_JvObjectPrefix</code>	Get G++ to allocate vtable pointer. Essentially, this is a pointer to a table of function pointers;
<code>Sync_info</code>	A pointer to <code>java.lang.Class</code>
<code>Next</code>	A pointer to <code>java.lang.Class</code>
<code>Name</code>	A pointer to an UTF8 constant string
<code>accflags</code>	Unsigned short, access flags for class
<code>Superclass</code>	A pointer to <code>java.lang.Class</code>
<code>loader</code>	A pointer to <code>java.lang.ClassLoader</code>
<code>Vtable</code>	A pointer to a table referred to the <code>Object</code> 's vtable
<code>State</code>	State of the class
<code>Thread</code>	A pointer to <code>java.lang.Thread</code>

Table 1: Some instance fields inserted by the GCJ compiler

Since our design goal was to implement JEM as far as possible in Java, we decided to overcome these restrictions by making the compiler-inserted fields visible and allowing additional fields be declared. As a result, this modification enables the JEM runtime to access and manipulate the meta-data generated by the GCJ compiler. The advantage of doing this is that the JEM could now be implemented in Java. Furthermore, this modification does not compromise the type safety of the Java language since these compiler-generated fields are considered private by default and only accessible by JEM.

4.3 Redirecting the GCJ Runtime Methods to JEM Methods

The GCJ runtime consists of a number of relatively high-level functions to provide memory management operations, thread and synchronization operations, type management operations, and class handling. These functions are implemented in C++, and some of the functions take arguments whose type cannot be represented as Java type. During the compilation process, the GCJ compiler inserts calls to these functions in the generated object code. In general, most of these internal functions are prefixed with `'_Jv_*'`. For example, `'_Jv_AllocObject'`, `'_Jv_NewArray'`, and `'_Jv_NewObjectArray'` are some of these internal runtime functions.

Since JEM has its own definition of these operations and we desire to implement these operations as far as possible in Java, we modified the GCJ compiler to redirect the inserted calls to the JEM methods. In order to minimize the number of modification made, we adopted the same naming convention. However, the JEM runtime support methods are all declared with either `package` or `private` visibility. Thus, avoiding any conflicts with the original library. Again, it is not possible to use these methods outside the runtime classes. Additionally, these changes are acceptable since they only affect the JEM runtime support without changing the semantic of the Java language.

4.4 JEM Low-level Support

Although most JEM methods were implemented in Java, a few were declared as native methods since their implementation depends on the targeted hardware. In such cases, explicit C++ and assembler code was used. These methods include ones for registers manipulation, context switching, and exception handling.

Sun's Java Development Kit (JDK 1.1) defined a programming interface known as Java Native Interface (JNI) for writing native methods in C or C++. However, the overhead of using the Java JNI is significant. For example, the native code needs to make two function calls to access a field in an object – one call to find the class and the other to access the field in that class. The GCJ compiler provides an alternate but more efficient native interface known as Cygnus Native Interface (CNI). The general idea of CNI is to make GNU Java compatible to GNU C++ and to provide hooks so the native code can access Java objects as naturally as C++ objects.

Since one of our design goals is provide an efficient Java micro-kernel, we implemented all the low-level support using the CNI approach. Although a reasonable performance improvement is expected, the downside is the resulting code is less portable. Specifically, we cannot use other C++ compilers to compile the native code since they will not understand the CNI interface semantics.

More work is still required on the JEM low-level support as the current JEM AHL APIs only works with Myrinet network card. A more generic AHL API is currently being developed to support a range of embedded devices.

5 Related Work

As mentioned earlier, the JEM is largely based on the design concepts of L4 micro-kernel. However, our implementation is being carried out in Java as opposed to the original assembly language and C. The JEM project is not only about micro-kernel design but also the ability to craft out a generic infrastructure for developing and testing embedded software on small devices. Additionally, this work aims to show that Java is a viable low-level programming language, and the JEM could be easily ported by carefully “drawing the red line” between Java code and hardware dependent code as well as by taking advantages of Java language construct.

Originally, the Java 1.0 and 1.1 specifications defined a single language and runtime structure. However, along with the success of Java, Sun Microsystems started to define a number of versions of the runtime targeted for different purposes. Currently, the Java 2 Micro Edition (J2ME) is the smallest runtime specification still supporting the full language. However, the J2ME specification is based on the assumption that the runtime environment is capable of executing Java class files, i.e., has an interpreter, JIT, or hardware support for byte code. However, in this work, we do not make such assumptions. On the contrary, we only assume that the target devices have limited memory space, and have neither a JVM nor hardware support for executing bytecode. Due to our assumptions and project goals, J2ME is not suitable.

To the best of our knowledge, the two research projects that are based on Java and provide direct user level access to system resources are Jaguar [18] and PANIC [19][20]. Jaguar accomplished this by translating the Java bytecode to machine code at runtime, while PANIC offloads protocol processing by splitting the protocol components across host and network interface. JEM is different in that it is not only aims to off-load protocol processing but also off-load application specific services. This requires JEM to address other design issues, mentioned earlier, than just by passing operating systems calls. However, JEM draws experiences from these two works (especially from Jaguar) in terms of providing support for user-level networking [21][22].

There are many research efforts on Java-based operating systems. The two projects that are closest to JEM goals are U-Net/SLE (Safe Language Environment) [23] and SPINE [24]. The U-Net/SLE project implemented a subset of the JVM for the LANai4; thereby allowing customized packet processing to occur on the network adaptor. Because of the limitation in both processor speed and memory size, the primary focus of their work was on the performance of the JVM. Their preliminary results indicated that the LANai4 processor lacked sufficient power to support the JVM interpretation efficiently. The SPINE project extended the ideas of SPIN [25][26] to the

network interface. Underlying SPINE is a Modula-3 runtime executing on the network interface. The common aspects of these two approaches are the use of a kernel-integrated compiler to generate safe-code inside the kernel space. This enabled them to reduce the address-space switch and thus reduce the IPC costs. However, the kernel level compiler might impose overhead on the kernel. On the other hand, JEM is based on the L4 address space abstraction, which does not incur the cost of an address space switch and a JVM could be implemented as a service.

Other Java-based operating systems efforts are J-Kernel [27], KaffeOS [28], Joust [29] and Sun's JavaOS [30]. J-Kernel is structured like a microkernel-based system with automatic stub generation for inter-task communication. KaffeOS is based on the Kaffe JVM that provides mechanisms to support a process model. Both of these systems require a separate JVM for each Java application, and all run in supervisor mode. The Joust project integrated a JVM into the Scout operating system [31]. Joust uses Scout's path abstraction to provide control over CPU time and network bandwidth. All these approaches are sometime also known as language-based extensible systems [32][33]. These systems are either limited by their monolithic structure or are built upon a full features operating system and JVM. Although J-Kernel and KaffeOS try to overcome the monolithic structure, their research is mainly concerned with introducing the traditional concept of a process and a "red line" [34]. Sun's original JavaOS was a standalone operating system written almost entirely in Java. It is described as a "first class operating system for Java applications", but appears to provide a single JVM with little separation between applications. It is being replaced by a new implementation termed "JavaOS for Business", that also only runs Java applications [30]. JavaOS for consumers is built on the Chorus micro-kernel in order to achieve real-time properties needed in embedded systems.

JEM takes a different approach by emphasizing flexibility and customizability. In particular, JEM takes the pure micro-kernel approach in which basic mechanisms are implemented only to support implementation of higher-level services, such as application protection with domain specific policy. In addition, like J-kernel and JavaOS, JEM is almost entirely written in Java and the use of native code is avoided wherever possible.

6 Conclusions

In this paper we have discussed the design and implementation of our Java Embedded Micro-kernel architecture. The purpose of this architecture is to provide application services on small devices, such as network cards or mobile phones.

It has been challenging implementing a Java micro-kernel capable of running directly on the Myrinet adaptor. The main implementation difficulties have come from unanticipated direction that of the intricate relationship between the GCJ compiler and its runtime support. Producing a micro-kernel with GNU C or C++ is clearly feasible and straightforward. However, producing a micro-kernel with the GCJ compiler is not so obvious. This is primarily due to the current GCJ compiler being very dependant on its runtime supports, i.e., `libgcj`. As a result, a good thorough knowledge of `libgcj` internals is required in order to eliminate the difficulties mentioned when creating a new runtime system. In our case, interfacing the JEM micro-kernel with its underlying hardware and overlying modules was challenging. We have learnt many lessons from this experience and anticipate that this familiarity will enable us to enhance the JEM further and allow us to port it to different hardware more easily.

Despite the difficulties, the idea of using Java as a low level language to implement a pure Java-based micro-kernel is feasible. We have overcome our difficulties in three ways:

- a) Use GCJ to produce native code instead of bytecode,
- b) Wire the GCJ runtime primitives back to the JEM runtime primitives, and

- c) Enable the JEM source code to access the meta-information generated by the compiler.

Additionally, our experience shows that writing a micro-kernel using an object-oriented language such as Java provides a number of benefits. For one, the resulting code is easier to understand and modify. The stricter language specification enables developers to enhance JEM with fewer possibilities to circumvent language level object abstractions that might break the underlying semantic framework. Additionally, it will be possible to port a large number of Java packages to the JEM with none or minimal changes, since the APIs we used are to a large extent compatible with the standard Java APIs. In short, our project shows that Java can be used as a viable language for low-level programming.

Currently, the JEM does not address the garbage collection issue. The JEM thread model is somewhat incompatible to the Java's thread package and the JEM event handling still requires more work. Once the micro-kernel has been stabilized, we plan to focus on benchmarking the various JEM components especially the communication performance. The implementation of various service modules for JEM is also warranted. We anticipate a stable JEM release in the early 2003.

Acknowledgements

The authors would like to thank Myricom for the loan of various network components, which are being used to prototype and test the JEM framework and code.

References

- [1] International Data Corporation (IDC), <http://www.windriver.com/press/html/jeode.html>
- [2] Myricom, <http://www.myri.com/>
- [3] Linux devices, <http://www.linuxdevices.com/articles/AT4936596231.html>
- [4] M. A. Baker and Hong Ong, A Java Embedded Micro-kernel Infrastructure, *Distributed Systems Group*, Technical Report 2002.04, 2002.
- [5] Andrew S. Tanenbaum, A Comparison of Three Microkernels, *The Journal of Supercomputing*, vol. 9, No ½, 1995.
- [6] A. Messer and T. Wilkinson, Components for Operating System Design, *5th International Workshop on Object Orientation in Operating Systems*, October 1996, Seattle, USA.
- [7] Real-time Java Specification, <http://www.rtg.org>
- [8] L. Carnahan and M. Ruak, Requirements for real time extensions for the Java platform, *RTJWG*, Technical Reports, September 1999.
- [9] H. McGhan and M. O'Connor, PicoJava: a direct execution engine for Java bytecode. *IEEE Computer*, 31(2), October 1998.
- [10] J. Liedtke, On Micro-kernel Construction, *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*. ACM Press, New York, pp. 2-11, 1995.
- [11] D. Hildebrand, An Architectural Overview of QNX, *Proceedings of the USENIX Workshop on Micro-kernel and Other Kernel Architecture*. Seattle, WA, 1992.
- [12] Michael Hohmuth, The Fiasco Kernel: Requirement Definition, *Technische Universita Dresden, Technical Report*, December 1998.
- [13] Virtual Interface (VI) Developer Forum, <http://www.vidf.org>
- [14] ANSI/VITA 26-1998, Myrinet-on-VME Protocol Specification, *VITA Standard Organization*. (<http://www.myri.com/open-specs/myri-vme-d11.pdf>)
- [15] M.A. Baker and D.B. Carpenter, MPJ: A Proposed Java Message-Passing API and Environment for High Performance Computing, *the 2nd Java Workshop at IPDPS 2000, Cancun, Mexico, LNCS*, Springer Verlag, Germany, pp 552 - 559, May 5th 2000, ISBN 3-540-67442-X.
- [16] M.A. Baker, D.B Carpenter, et al., mpiJava: An Object-Oriented Java interface to MPI, the *1st Java Workshop at the 13th IPPS & 10th SPDP Conference*, Puerto Rico, April 1999, LNCS, Springer Verlag, Heidelberg, Germany, ISBN 3-540-65831-9.
- [17] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox: MPJ: MPI-like message passing for Java. *Concurrency - Practice and Experience* 12(11): 1019-1038 (2000).

- [18] M. Welsh and D. Culler, Jaguar: Enabling Efficient Communication and I/O in Java, *Concurrency: Practice and Experience*, vol. 12, pp. 519-538, Special issue on Java for High –Performance Application, Dec. 1999.
- [19] PANIC, http://www.cs.tamu.edu/faculty/bettati/panic_overview.html
- [20] C. Beauduy and R. Bettati, Protocols Aboard Network Interface Cards (PANIC), *Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems*, Cambridge Massachusetts, USA, November 3-6, 1999
- [21] M. Welsh, A. Basu, and T. von Eicken, Incorporating Memory Management into User-Level Network Interfaces, *In Proceedings of Hot Interconnects V*, Stanford, August 21-23, 1997
- [22] M. Welsh, Safe and Efficient Hardware Specialization of Java Applications, *UC Berkeley Technical Report*, May 2000.
- [23] M. Welsh, D. Oppenheimer, and D. Culler, U-Net/SLE: A Java-based User-Customizable Virtual Network Interface, *In Scientific Programming*, Vol. 7, No. 2, 1999, Special Issue on High Performance Java Compilation and Runtime Issues.
- [24] M.E. Fiuczynski and B. N. Bershad, SPINE – A Safe Programmable and Integrated Network Environment, *In Proceedings of the 8th ACM SIGOPS European Workshops*, 1998
- [25] B.N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, S. Eggers, and C. Chambers, Extensibility, Safety and Performance in the SPIN Operating System, *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, 1995
- [26] N. Wirth and J. Gutknecht, Project Oberon, *ACM Press*, New York, NY, 1992
- [27] C. Hawblitzel, T. von Eicken, A case for language-based protection, *Technical Report TR-98-1670, Cornell University*, March 1998.
- [28] G. Back, W. C. Hsieh, J. Lepreau, Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. *Proceedings of the 4th OSDI*, pp. 333-346, Oct 2000.
- [29] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheck. Experiences building a communication-oriented JavaOS. *Software--Practice and Experience*, Vol. 30, Issue 10, April 2000.
- [30] JavaOS for Business, <http://developer.java.sun.com/developer/products/JavaOS/>
- [31] A.B. Montz, D. Mosberger, S.W. O'Malley, L.L. Peterson, and T.A. Proebsting, Scout: A Communications-Oriented Operating System. *Hot OS*, May 1995.
- [32] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell, Pilot: An Operating system for a personal computer, *Communication of ACM*, 23(2), pp. 81-92, 1980.
- [33] D. C. Swinehart, P. T. Zellweger, R. J. Beach, and R. B. Haggmann, A structural view of the cedar programming environment, *ACM Transactions on Programming Languages and Systems*, 8(4), pp. 419-490, Oct. 1986.
- [34] D. R. Cheriton. Low and High Risk Operating System Architectures. *Proceedings of OSDI*, pp. 197, Nov. 1994.